PATTERN-MATCHING and LAMBDA FUNCTION THEORY

William Bricken
April 1992

The Appendix contains a summary of lambda calculus transformation rules.

THE GENERAL IDEA

Select a robust and useful match-and-substitute (M&S) domain to test ideas and performance. Use recursive function theory. Enhance expressibility afterwards.

Mathematical functions work by substituting values for variables, then doing arithmetic simplification. It's convenient for a language to permit functions to act on other functions. The central control issue is what functions to evaluate when. We are heading toward simple substitution rules with a simple control structure.

So there are three focal areas:

#1 subdomains that give us increasing expressibility without changing the computational metaphor
#2 control structures around M&S which tell us which target for matching to select when
#3 actually doing a specific M&S

LAMBDA CALCULUS MATHEMATICAL APPROACH

Constants
literal, exactly as they are written.
Expressions
E when more than one {E1, E2,...}
Variables
{x1, x2,...}
Functions
{f1, f2,...}

ATOMS are constants and variables

Syntax

<Lexpression> ::= <variable> | <constant> | <application> | <abstraction>

<application> ::= <Lexpression> <Lexpression>

<abstraction> ::= <variable> < <Lexpression> >

Compositional operations

	Conventional notation	Boundary notation
APPLY:	E1 E2	(E1)E2
ABSTRACT:	Lx.E	x< E >

The boundary notation for ABSTRACTION is used herein.

APPLY

APPLY means substitute a value for a variable in a functional expression. I'll use lisp-like [subst x y E] for "substitute x for y in expression E"

E1 E2 ==> [substitute E2 for variable specified by xn<...> around E1]

The BETA reduction rule of lambda calculus looks like:

x<E1> E2 ==> [subst E2 x E1] ==> new E1

APPLY is *left-associative*, so that E1 E2 E3 = (((E1) E2) E3).

The important implication is that complex expressions group to the left:

(first (E1 E2 ... En)) ==> (E1 E2 ... En-1)
(second (E1 E2 ... En)) ==> En

ABSTRACT

ABSTRACT means to remove the dependence on the variable in a functional expression, to generalize the function to a domain. It will go away completely later.

BETA RULE

Idea: Substitution as function evaluation, as computation.

These will all disappear in the combinator approach which follows, they are included to illustrate how to think about substitution in the context of application and abstraction.

This is the sole mechanism of lambda computation and the way computation or reduction proceeds. What is happening here is that the abstraction x<E1> is APPLIED to the right-most expression E2.

```
B1.  x<x> E          ==>  E

B2.  x<y> E          ==>  y

B3.  (x< x< E1>>)E2  ==>  x< E1 >

B4.  x1< x2<E1> > E  ==>  x2< x1<E1> E >

B5.  x< E1 E2 > E    ==>  x<E1> E (x<E2> E)
```

read B4 as [subst E for x1 in (x2<E1>)] = x2< [subst E for x1 in E1] >

read B5 as [subst E for x in (E1 E2)] = [subst E x E1] [subst E x E2]

Examples

```
f1(x) = x + 1           is    f1 =  x< +1 x >

f1(3)  ==>  x< +1 x > 3
       ==>  [substitute 3 for x in (+1 x)]
       ==>                         (+1 3)
       ==>                          4

f2(x, y) = 2*(x + y)    is    f2 =  x1< x2< *2 (+ x1 x2) >>

f2(3, 4)  ==>  x1< x2< *2 (+ x1 x2) >> 3 4
          ==>  x2< x1< *2 (+ x1 x2)> 3 > 4
          ==>  x2< [subst 3 for x1 in (*2 (+ x1 x2))] > 4
          ==>  x2<                     *2 (+  3 x2)   > 4
          ==>  [subst 4 for x2 in (*2 (+3 x2))]
          ==>                     (*2 (+3  4))
          ==>                     (*2  7     )
          ==>                      14
```

The first step above is rule B4, which migrates expressions through multiple abstraction boundaries.

Note that operators apply immediately to constants, ie (+) 3 ==> +3
Here, the "+3" is an "add 3" operator. All functions in this approach apply to one and only one argument.

EVALUATION

In general, there are two principled ways to evaluate an expression:

NORMAL ORDER: apply outermost leftmost lambda variable
(demand-driven, call-by-need, lazy)

APPLICATIVE ORDER: apply innermost leftmost lambda variable
(data-driven, call-by-name, eager)

The strength is that it doesn't matter which order these substitutions are applied. And they can be applied asynchronously in parallel.

Normal order reduction is SAFE, in that it produces a normal form (when such a form exists). Applicative order is reputed to be more efficient, but can fall into infinite loops when dealing with infinite data structures. The situation is analogous to breadth-first vs depth-first search of trees.

It is possible to standardize expressions so that either control approach is trivial to unfold.

Examples

```
f1(f2(3,4)) = f2(3,4) + 1 = 2*(x + y) + 1

f1(f2(3,4))    is    f1 f2 3 4
               is    x< +1 x >  x1< x2< *2 (+ x1 x2) >> 3 4
```

Normal order

```
      x< +1 x > x1< x2< *2 (+ x1 x2) >> 3 4

==> [subst (x1< x2< *2 (+ x1 x2) >>) for x in (+1 x)] 3 4
==> (+1 x1< x2< *2 (+ x1 x2) >>)                     3 4
==> (+1 x2< x1< *2 (+ x1 x2) > 3 > 4
==> (+1         [subst 4 for x2 in (x1< *2 (+ x1 x2) > 3)] )
==> (+1                            (x1< *2 (+ x1  4) > 3)  )
==> (+1 [subst 3 for x1 in (*2 (+ x1 4))] )
==> (+1                    (*2 (+  3 4))
==> (+1                    (*2  7      )
==> (+1                     14         )
==>  15
```

Applicative order

```
      x< +1 x > x1< x2< *2 (+ x1 x2) >> 3 4

==> x< +1 x > x2< x1< *2 (+ x1 x2) > 3 > 4                    using B4
==> x< +1 x > x2< [subst 3 for x1 in (*2 (+ x1 x2))] > 4
==> x< +1 x > x2<                    (*2 (+  3 x2))  > 4
==> x< +1 x >     [subst 4 for x2 in (*2 (+  3 x2))]
==> x< +1 x >                        (*2 (+  3  4))
==> x< +1 x >                        (*2  7       )
==> x< +1 x >                         14
==>    [subst 14 for x in (+1  x)]
==>                       (+1 14)
==>                        15
```

Mixed order

```
      x< +1 x > x1< x2< *2 (+ x1 x2) >> 3 4

==> x< +1 x > x2< x1< *2 (+ x1 x2) > 3 > 4
==> x< +1 x > [subst 3 for x1 in (x2< *2 (+ x1 x2) >) ] > 4
==> x< +1 x >                    (x2< *2 (+  3 x2) >)   > 4
==> [subst (x2< *2 (+ 3 x2) >) for x in (+1 x)]           4
==>                                  (+1 (x2< *2 (+ 3 x2) >)) 4
==> (+1 (x2< *2 (+ 3 x2) > 4))
==> (+1 [subst 4 for x2 in (*2 (+ 3 x2))] )
==> (+1                    (*2 (+ 3  4))  )
==> (+1                    (*2  7      )  )
==> (+1                     14            )
==>  15
```

FACTORIAL EXAMPLE

Applicative order

```
(defun fac (n) (if (= n 0) 1 (* n (fac (-1 n)))))

(fac 3) ==> [ subst  3  n  (if (= n 0) 1 (* n (fac (-1 n)))) ]
        ==>                (if (= 3 0) 1 (* 3 (fac (-1 3))))
        ==>                (if  F      1 (* 3 (fac  2    )))
        ==>                                (* 3 (fac  2    ))
        ==> (* 3 [ subst  2  n  (if (= n 0) 1 (* n (fac (-1 n)))) ] )
        ==> (* 3                (if (= 2 0) 1 (* 2 (fac (-1 2))))   )
        ==> (* 3                (if  F      1 (* 2 (fac  1    )))   )
        ==> (* 3                              (* 2 (fac  1    ))    )
        ==> (* 3 (* 2 [ subst 1 n (if (= n 0) 1 (* n (fac (-1 n)))) ] ))
        ==> (* 3 (* 2             (if (= 1 0) 1 (* 1 (fac (-1 1))))   ))
        ==> (* 3 (* 2             (if  F      1 (* 1 (fac  0    )))   ))
        ==> (* 3 (* 2                           (* 1 (fac  0    )))   ))
        ==> (* 3 (* 2 (* 1 [ subst 0 n (if (= n 0) 1 (* n (fac (-1 n)))) ] )))
        ==> (* 3 (* 2 (* 1             (if (= 0 0) 1 (* 0 (fac (-1 0))))   )))
        ==> (* 3 (* 2 (* 1             (if  T      1 (* 0 (fac  -1   )))   )))
        ==> (* 3 (* 2 (* 1                           1                      )))
        ==> (* 3 (* 2 (* 1                           1                      )))
        ==> (* 3 (* 2  1                                                    ))
        ==> (* 3  2                                                          )
        ==>  6
```

Normal order

```
(fac 3)  ==>  [ subst  3  n  (if (=0 n) 1 (* n (fac (-1 n)))) ]

      (if (=0 3) 1
          (* 3 [subst  2  n  (if (=0 n) 1
                                 (* n (fac (-1 n)))) ]) ]

      (if (=0 3) 1
          (* 3 (if (=0 2) 1
                   (* 2 [subst 1 n (if (=0 n) 1 (* n (fac (-1 n)))) ] )) ))

      (if (=0 3) 1
          (* 3 (if (=0 2) 1
                   (* 2 (if (=0 1) 1
                            (* 1 [subst 0 n (if (=0 n) 1
                                                (* n (fac -1 n))) ] )) )) ))

      (if (=0 3) 1
          (* 3 (if (=0 2) 1
                   (* 2 (if (=0 1) 1
                            (* 1 (if (=0 0) 1
                                     (* 0 (fac -1 0))) )) )) ))

      (if (=0 3) 1
          (* 3 (if (=0 2) 1
                   (* 2 (if (=0 1) 1
                            (* 1 1))) )) ))

      (if (=0 3) 1
          (* 3 (if (=0 2) 1
                   (* 2 (if (=0 1) 1
                            1)) )) ))

      (if (=0 3) 1
          (* 3 (if (=0 2) 1
                   (* 2 1)) ))

      (if (=0 3) 1
          (* 3 (if (=0 2) 1
                   2) ))

      (if (=0 3) 1
          (* 3 2))

      (if (=0 3) 1 6)

                   6
```

COMBINATORS, A PATTERN MATCHING LANGUAGE FOR FUNCTIONS

These are expressions that do not interact with their context (ie independent of other expressions). They are convenient macros that can be expanded for substitution, or they can be reduced using their own rules.

Turns out, we can use combinators to eliminate the idea of ABSTRACTION. The conversion:

```
x< const >       ==>   K const

x< combinator >  ==>   K combinator

x< variable >    ==>   K variable
                 ==>   I                    when variable = x

x< E1 E2 >       ==>   S  x<E1>  x<E2>
```

S, K, and I are combinator "functions" that basically encapsulate control structure. There are other combinators which increase efficiency of expansion.

Function Evaluation via Substitution

So here are a set of M&S rules to achieve functional evaluation. Some of the rules can be seen as compiling rules, to get normal functions into combinator form, the rest are the rules that achieve computation.

The input language is LISP, so we will have to add the LIST domain (car, cdr, cons). I assume a facility to reduce arithmetic expressions. We will also need a symbol table for testing valid labels:

```
(atom "x") ==> True     when "x" is a member of the active symbol table
```

LISP to LAMBDA

```
(defun f1 (x) body)  ==>  f1 = x< body >

(f1 constant)        ==>  x< body > constant

(f1 (f2 whatever))   ==>  f1 f2 whatever
```

"f1 = ..." means
"pattern match on f1 as a label and substitute its definition"

```
      [ subst rhs for f1 in body ]
```

LAMBDA to COMBINATOR

```
x< ground >       ==>  K ground

x< x >            ==>  I

x< E1 E2 >        ==>  S  x<E1>  x<E2>
```

Combinator reduction

```
S K K             ==>  I

I E               ==>  E

K E1 E2           ==>  E1

S E1 E2 E3        ==>  E1 E3 (E2 E3)

S (K E1) (K E2)   ==>  K (E1 E2)

S (K E1) I        ==>  E1
```

Combinator optimization

```
W E1 E2           ==>  E1 E2 E2

S (K E1) E2       ==>  B E1 E2

B E1 E2 E3        ==>  E1 (E2 E3)

S E1 (K E2)       ==>  C E1 E2

C E1 E2 E3        ==>  E1 E3 E2

S (B E1 E2) E3    ==>  SP E1 E2 E3

SP E1 E2 E3 E4    ==>  E1 (E2 E4) (E3 E4)

B (E1 E2) E3      ==>  BP E1 E2 E3

BP E1 E2 E3 E4    ==>  E1 (E2 (E3 E4))

C (B E1 E2) E3    ==>  CP E1 E2 E3

CP E1 E2 E3 E4    ==>  E1 (E2 E4) E3
```

Recursion

```
Y E              ==>  E (Y E)
```

About Match and Substitute

We need to do [subst E1 for x in E2], and the resultant simplification of arithmetic in the (content-addressable) memory. The issue is that E1 and E2 are expressions of arbitrary size.

"x" may show up many times in E2, but we don't want to substitute the same thing many times before evaluation. For example, avoid

```
x< + x x > hairy-function  ==>  (+ hairy-function hairy-function)
```

Combinators do this by pattern matching duplicate expressions:

```
W E1 E2  ==>   E1 E2 E2
```

When we have some complex expression, combinator rules are completely independent, and can be applied in parallel.

So the issue is how to:

```
-- match the left-hand-side of each rule
-- substitute the right hand side for the configuration, and then
-- simplify the arithmetic structures that occur.
```

These transformations are not the same as those generated by logic or algebra, but the issues will remain the same.

It's easy to add new domains (lists, sets, records, etc).

The control issue is now: Which rules should I select to try for reduction? There are principled approaches and there are random approaches. The essential point is that *it doesn't matter*, the system will still work, although it might be more inefficient with random reductions.

Actually, the reduction sequence will depend on the objective. Normalization of expressions can make certain rules more likely to trigger.

Note that an expression in this language can be interpreted as a tree. See below.

SIMPLE EXAMPLE

```
f3 = (x1 + x2)      ==>      f3 = x1< x2< + x1 x2 >>
```

F3 expressed as combinators

```
x1< S x2<+ x1> x2<x2> >

x1< S (S x2<+> x2<x1>) I >

x1< S (S (K +) (K x1)) I >

S x1< S (S (K +) (K x1)) > x1<I>

S (S x1<S> x1<S (K +) (K x1)>) (K I)

S (S (K S) (S x1<S (K +)> x1<K x1>)) (K I)

S (S (K S) (S (S x1<S> x1<K +>) (S x1<K> x1<x1>))) (K I)

S (S (K S) (S (S (K S) (S x1<K> x1<+>)) (S (K K) I))) (K I)

S (S (K S) (S (S (K S) (S (K K) (K +))) (S (K K) I))) (K I)
```

Yes this is a complex way to add two numbers. Using simplifying rules when possible, from the top:

```
x1< S x2<+ x1> x2<x2> >

x1< S (S x2<+> x2<x1>) I >

x1< S (S (K +) (K x1)) I >

x1< S (K (+ x1))       I >

x1< (+ x1)               >

S x1<+> x1<x1>

S (K +) I

+
```

So in this case, we can toss the *concept of variables* from the plus operator. "+" becomes a generalized accumulator upon application.

FACTORIAL EXAMPLE

Three versions of factorial follow:

A. The Y combinator only, combined with conventional Lx. abstraction.
B. All combinators, combined with boundary x< E > abstraction.
C. All combinators, with no abstraction.

Factorial version A (Y combinator, Lx. notation)

```
      (fac)n = if n=0 then 1 else ((*)n)(fac)(-1)n

            where =0  is  Ln.((n)(T)F)T

                  -1  is  Ln.(((n)Lp.Lz.((z)(+1)(p)T)(p)T)Lz.((z)=0)=0)F

                   *  is  Lm.Ln.Lf.(m)(n)f

                  +1  is  Ln.Lf.Lx.(f)((n)f)x


      FAC =  (Y)Lf.Ln.(((=0)n)1)((*)n)(f)(-1)n

(FAC)2 ==> ( (Y)Lf.Ln.(((=0)n)1)((*)n)(f)(-1)n  ) 2

       ==>   (Y)Lf.   (((=0)2)1)((*)2)(f)(-1)2

       ==>   (Y)Lf.   ((  F  )1)( *2 )(f) 1

       ==>   (Y)Lf.             ( *2 )(f) 1

       ==>                      ( *2 )( (Y)Lf.Ln.(((=0)n)1)((*)n)(f)(-1)n  )1

       ==>                      ( *2 )  (Y)Lf.   (((=0)1)1)((*)1)(f)(-1)1

       ==>                      ( *2 )  (Y)Lf.   ((  F  )1)( *1 )(f)(-1)1

       ==>                      ( *2 )  (Y)Lf.             ( *1 )(f) 0

       ==>  (*2) (*1) ((Y)Lf.Ln.(((=0)n)1)((*)n)(f)(-1)n)0

       ==>  (*2) (*1)  (Y)Lf.   (((=0)0)1)((*)0)(f)(-1)0

      ==>   (*2) (*1)  (Y)Lf.   ((  T  )1)( *0 )(f) -1

      ==>   (*2) (*1)  (Y)Lf.           1

      ==>   (*2) (*1)                   1        ==>     2
```

Factorial version B (all combinators, x< E > notation)

Rules for the COND combinator:

```
COND T E2 E3 ==> E2
COND F E2 E3 ==> E3
```

Alternatively, the logic of could be expressed in terms of primitives {TRUE, NOT, AND}. Note that the K combinator means TRUE in logic.

```
     FAC = Y f< n< COND (=0 n) 1 (* n (f (-1 n))) >>

FAC 2 ==>  Y f< n< COND (=0 n) 1 (* n (f (-1 n))) >>  2

       ==>  f< n< COND (=0 n) 1 (* n (f (-1 n))) >> (Y FAC) 2

       ==>  n< COND (=0 n) 1 (* n ((Y FAC) (-1 n))) > 2

       ==>  COND (=0 2) 1 (* 2 ((Y FAC) (-1 2)))

       ==>  COND  F 1 (*2 ((Y FAC) 1))

       ==>  (*2 ((Y FAC) 1))

       ==>  *2 ((Y f< n< COND (=0 n) 1 (* n (f (-1 n))) >>) 1)

       ==>  *2 ((f< n< COND (=0 n) 1 (* n (f (-1 n))) >> (Y FAC)) 1)

       ==>  *2 (n< COND (=0 n) 1 (* n ((Y FAC) (-1 n))) > 1)

       ==>  *2 (COND (=0 1) 1 (* 1 ((Y FAC) (-1 1))))

       ==>  *2 (COND  F 1 (*1 ((Y FAC) 0)))

       ==>  *2 (*1 ((Y FAC) 0))

       ==>  *2 *1 ((Y f< n< COND (=0 n) 1 (* n (f (-1 n))) >>) 0)

       ==>  *2 ((f< n< COND (=0 n) 1 (* n (f (-1 n))) >> (Y FAC)) 0)

       ==>  *2 (n< COND (=0 n) 1 (* n ((Y FAC) (-1 n))) > 0

       ==>  *2 (COND (=0 0) 1 (* 0 ((Y FAC) (-1 0))))

       ==>  *2 (COND T 1 (* 0 ((Y FAC) -1)))

       ==>  *2 1  ==>  2
```

The above example uses parallel reduction on arithmetic expressions, but the expansion of FAC is a single computational thread. In an environment where many function threads are being expanded, this one can be in parallel with others.

Factorial version C (all combinators, no abstraction)

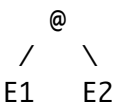
The combinator representation of FAC is:

```
     FAC = (S (C (B COND =0) 1) (S * (B FAC -1)))

         = (S (CP COND =0 1) (S * (B FAC -1)))

FAC 2 ==>  (S (CP COND =0 1) (S * (B FAC -1))) 2

       ==>  (CP COND =0 1) 2 ((S * (B FAC -1)) 2)

       ==>  COND (=0 2) 1 (S * (B FAC -1) 2)

       ==>  COND F 1 (S * (B FAC -1) 2)

       ==>  S * (B FAC -1) 2          this could be in parallel with above

       ==> * 2 ((B FAC -1) 2)

       ==> * 2 (FAC (-1 2))

       ==> *2 (FAC 1)

       ==> *2 (S (CP COND =0 1) (S * (B FAC -1)) 1)

       ==> *2 (CP COND =0 1) 1 ((S * (B FAC -1)) 1)

       ==> *2 COND (=0 1) 1 (* 1 ((B FAC -1) 1))              parallel

       ==> *2 COND F 1 (*1 (FAC (-1 1)))

       ==> *2 (*1 (FAC 0))

       ==> *2 *1 (S (CP COND =0 1) (S * (B FAC -1)) 0)

       ==> *2 (CP COND =0 1) 0 ((S * (B FAC -1)) 0)

       ==> *2 COND (=0 0) 1 (* 0 ((B FAC -1) 0))

       ==> *2 COND T 1 (*0 (FAC (-1 0)))

       ==> *2 1  ==> 2
```

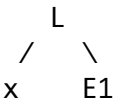
GRAPHS QUICKLY

Note that the whole graph formalism is simply using pointers rather than text for substitution. This can control the size of the substitution expression.

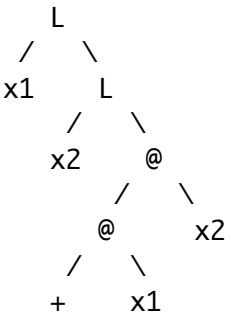
E1 E2 is APPLY E1 to E2



x< E1> is ABSTRACT x

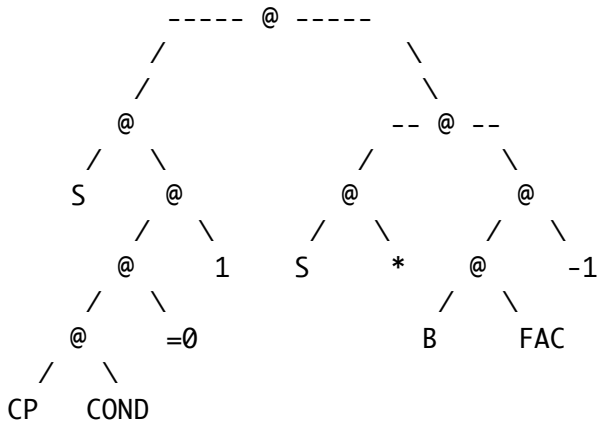


So f3 = x1< x2< + x1 x2 >>



Combinator graph for FAC

FAC = (S (CP COND =0 1) (S * (B FAC -1)))



THE RESEARCH QUESTIONS

How fast can we do naive (brute-force) M&S on combinator expressions? Compare to standard LISP for speed.

How many rules to use for reduction? Less rules on longer expressions or more rules on more succinct expressions? Using S-K-I only (6 rules) causes exponential growth of expression size relative to number of abstractions. Adding W-B-C (5 more rules)reduces expression size to quadratic. Adding SP-BP-CP (6 more rules) reduces expression size to linear.

Use graph/pointer representation or text/string representation?

How to partition rule-bases, so we get "apply group I rules until done, then group II, ..."

How to say "apply this set of rules until no more reductions".

APPENDIX

LAMBDA CALCULUS SUMMARY

SYNTAX

<Lexpression> ::= <variable> | <constant> | <application> | <abstraction>

<application> ::= (<Lexpression>) <Lexpression>

<abstraction> ::= L<variable> . <Lexpression>

BETA APPLICATION RULES

Assume unique variable names {x1 .. xn}

Lx. = [subst the-second-E-on-the-right for x in the-first-E-on-the-right]

(Lx.E1)E = APPLY E1 to E, by substituting for the containing lambda

The left-hand-side specifies a pattern. The right hand side is the transform.

B0. (Lx.E1)E2 ==> [subst E2 x E1]

B1. (Lx.x)E ==> E

B2. (Lx.y)E ==> y y is a ground and y =/= x

B3. (Lx.Lx.E1)E2 ==> Lx.E1

B4. (Lx.Ly.E1)E2 ==> Ly.(Lx.E1)E2

B5. (Lx.(E1)E2)E3 ==> ((Lx.E1)E3)(Lx.E2)E3

ALPHA SUBSTITUTION RULES

Unique variable names make the ALPHA RULES never used.
A4a and A4b formally avoid naming conflicts, and are *never used* in practice.

```
A1.   [subst E x x]       = E

A2.   [subst E x y]       = y

A3.   [subst z x Lx.E ]   = Lz.[subst z x E]

A3'.  [subst E2 x Lx.E1]  = Lx.E1         Lx binds x internally in E1

A4a.  [subst E2 x Ly.E1]  = Ly.[subst E2 x E1]        where x =/= y

                          onlyif  not ( x free in E1  and  y free in E2 )

A4b.  [subst E2 x Ly.E1]  = Lz.[subst E2 x [subst z y E1]]

                          onlyif      ( x free in E1  and  y free in E2 )

A5.   [subst E3 x (E1)E2] = ( [subst E3 x E1] ) [subst E3 x E2]
```

RULES SUMMARY

```
ALPHA:   Lx.E        ==>  Lz.[subst z x E]  for new z

ALPHA:   Lx.E        ==>  Lz.(Lx.E) z

BETA:  ( Lx.E1 )E2   ==>  [subst E2 x E1]
```

CONTROL THEORY

Applicative: substitute the left-most inner-most Lambda

```
f = Lx.Ly.((+)x)y   ==>   Lx.((+)x)[subst y-value y f]
```

Normal: substitute the left-most outer-most Lambda

```
f = Lx.Ly.((+)x)y   ==>   Ly.((+)[subst x-value x f])y
```

COMBINATORS

These are expressions that do not interact with their context (ie independent of other expressions). They are convenient macros that can be expanded for BETA application, or they can be reduced using their own rules.

```
K = Lx.constant                                   constant K

I = Lx.x                                          the identity I

C = Lf.Lg.Lx.(f)(g) x                             function composition C

S = Lx.Ly.Lz.((x)z)(y)z                           distribution S

      I = ((S)K)K
```

Logic

```
T = Lx.Ly.x                                       true T

F = Lx.Ly.y                                       false F

N = Lx.((x)F)T                                    not N
```

Recursion

```
Y = Ly.(Lx.(y)(x)x)Lx.(y)(x)x                     fixed-point combinator Y

WHILE = Lp.Lf.Lx.(((p)x)(((WHILE)p)f)(f)x)x

      = (Y)Lw.Lp.Lf.Lx.(((p)x)(((w)p)f)(f)x)x
```

Reduction Rules

```
(I)E           ==>  E

(((C)E1)E2)E   ==>  (E1)(E2)E

(((S)E1)E2)E3  ==>  ((E1)E3)(E2)E3

((T)E1)E2      ==>  E1

((F)E1)E2      ==>  E2

(Y)E           ==>  (E)(Y)E
```

Eliminating Abstraction with Combinators

```
      Lx.<const>       ==>  (T)<const>

      Lx.<combinator>  ==>  (T)<combinator>

      Lx.<variable>    ==>   I                      when <variable> = x
                       ==>  (T)<variable>          when not (<variable> = x)

      Lx.(E1)E2        ==>  ((S)Lx.E1)Lx.E2
```

Arithmetic Domain

```
      +1 = Ln.Lf.Lx.(f)((n)f)x

      +  = Lm.Ln.Lf.Lx.((m)f)((n)f)x

      *  = Lm.Ln.Lf.(m)(n)f

      =0 = Ln.((n)(T)F)T
```

List Domain

```
      HEAD = Lx.(x)T
      TAIL = Lx.(x)F
      CONS = Lx.Ly.Lz.((z)x)y

            (HEAD)[]                ==>  []
            (HEAD)[E1,...,En]       ==>  E1

            (TAIL)[]                ==>  []
            (TAIL)[E1,...,En]       ==>  [E2,...,En]

            ((CONS)E)[]             ==>  [E]
            ((CONS)E)[E1,...,En]    ==>  [E,E1,...,En]

            (NULL)[]                ==>  T
            (NULL)[E1,...,En]       ==>  F

            ((MAP)f)[]              ==>  []
            ((MAP)f)[E1,...,En]     ==>  [(f)E1,...,(f)En]

ALPHA:      [subst z x [E1,...,En]]  ==>  [[subst z x E1],...,[subst z x En]]

BETA:       ([E1,...,En])E          ==>  [(E1)E,...,(En)E]
            Lx.[E1,...,En]          ==>  [Lx.E1,...,Lx.En]
```